1992

Reservable Transactions: An Approach for Reliable Multidatabase Transaction Management

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Jin Jing

James G. Mullen

Jamshid Sharif-Askary

Report Number:
92-012

RESERVABLE TRANSACTIONS: AN APPROACH FOR RELIABLE MULTIDATABASE TRANSACTION MANAGEMENT

Ahmed K. Elmagarmid
Jamshid Sharif-Askary
Jin Jing
James G. Mullen

CSD-TR-92-012
February 1992

Reservable Transactions: An Approach for Reliable Multidatabase Transaction Management*

Ahmed K. Elmagarmid
Jin Jing
James G. Mullen
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Jamshid Sharif-Askary
Harris Corporation
P.O. Box 430
Melbourne, FL 32902

Keywords: multidatabase systems, federated database systems, heterogeneous distributed database systems, commitment protocols, atomic commitment, transaction management

Abstract

Atomically committing global transactions in a multidatabase system is difficult because of local system autonomy. Traditional transaction management methods, such as two phase commit (2PC), are often not possible, since many database systems do not support a (visible) prepare-to-commit state. And, even where the local database systems of a multidatabase do provide such support, the potential blocking and long delays of 2PC can severly degrade local execution autonomy. Transaction models that use compensation (such as Sagas), can avoid some of these problems, however these methods are only appropriate for certain cases. This paper presents *reservable transactions*, which contain a reservation phase that utilizes data semantics to ensure that a transaction can commit, without completely blocking the data that the transaction accesses. The method presented can handle cases where compensation is not applicable. In addition, the reservation concept can be used to increase the cases where compensation *is* applicable.

*This research is funded by the Indiana Corporation for Science and Technology (CST), a PYI Award from NSF under grant IRI-8857952, a Graduate Student Researcher Program (GSRP) grant from NASA, and grants from the AT&T Foundation, Tektronix, SERC, Mobil Oil and Bell Northern Research.

1 Introduction

Multidatabase systems combine autonomous and heterogeneous component (or local) database systems into a global database system. In multidatabase systems, global transactions are divided into subtransactions, with one subtransaction per local system that the global transaction accesses. Local transactions may be executed at each local database system. A conceptual view of a multidatabase system is shown in Figure 1. A global transaction G_i, and its decomposition into subtransactions $G_{i,1}$, $G_{i,2}$, ..., $G_{i,n}$ is shown. Also, a local transaction $L_{j,1}$ that executes at $LDBS_1$, and a local transaction $L_{k,n}$ that executes at $LDBS_n$, are shown.

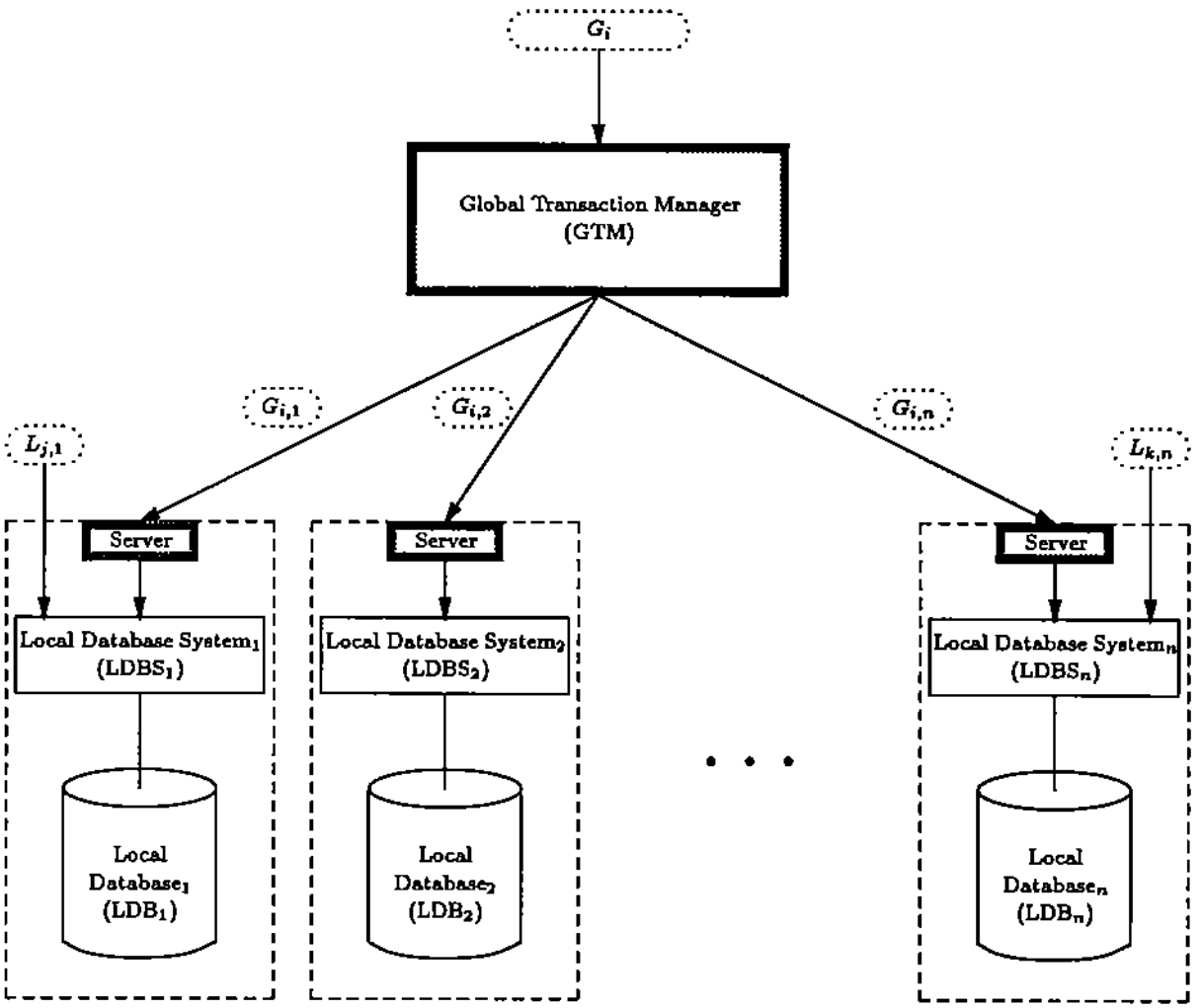

Figure 1: Conceptual Multidatabase Architecture.

One of the most difficult problems with implementing reliable transaction management in multidatabase systems is the problem of atomic commitment of global transactions. That is, ensuring that if any of the effects of a global transaction are executed, then all of the effects will be executed. In fact, in [MEK92] it has been shown that it is *impossible* to do in general without violating local autonomy. The two phase commit protocol (2PC) [LS76, Gra78] has been used for tightly coupled distributed database systems. However, there are at least two problems with applying 2PC to the multidatabase case. First, current database systems do not generally provide the (visible) prepare-to-commit state which is necessary to implement 2PC. Second, even if database systems of the future generally do provide the prepare-to-commit state, 2PC can severly violate local execution autonomy. Blocking can often occur with 2PC, so it becomes possible for a remote system to, in effect, lock another system's data for indefinite periods of time.

One approach to handling the atomic commitment problem, that avoids the blocking behavior of two phase commitment, is to use compensating transactions [GM83], such as used in Sagas [GMS87] or the (multidatabase) Flex transaction model [ELLR90]. Compensating transactions can undo the effects of a committed transaction, so that subtransactions of a transaction can be committed independently. Compensation can be considered an optimistic approach, in the sense that one goes ahead and commits subtransactions, with the hope that the entire global transaction will commit. If it does not, then the committed subtransactions can always be undone. Compensation helps support long-lived transactions, since data is not blocked until the entire transaction commits. In addition, compensation does not require a visible prepare-to-commit state. However, the applicability of compensation depends on the semantics of the data/transactions involved. So, compensation will not work for all cases. If the operations that can be performed on a data item are not commutative (i.e. if the order of execution matters), then compensation will not work. Also, if the effects of a subtransaction correspond to a real world event (e.g. launching a missile), then the effects might not be compensatable.

We present a transaction management approach called *reservable transactions*. Roughly speaking, whereas compensation could be termed an optimistic approach, reservable transactions could be termed a pessimistic approach. A reservable transaction is a global transaction that has a reservation phase that is executed first, and that attempts to ensure that all of the subtransactions of the global transaction can commit. In effect, this is what 2PC does, however, reservable transaction use the semantics of the data/transactions involved so that the reservation stage does not totally block the data items that will be accessed by the global transaction. Only "harmful" transactions will be blocked. In addition, reservable transactions do not require local database systems to support a visible prepare-to-commit

state.

The remainder of this paper is organized as follows. In Section 2, we present the basic idea of reservable transactions, and we present example reservable transactions and discuss the benefits of the reservable transaction approach. Section 3 presents a formal model and proof of correctness of the reservation approach. Section 4 discusses how to implement reservable transactions and discusses the effects on of this implementation on local system autonomy. Section 5 presents our conclusions.

2 Reservable Transactions

Conceptually, a reservable transaction is a global transaction that has a reservation phase. The goal of the reservation phase is to guarantee that the global transaction can commit. Due to failures, it is impossible to guarantee that the first attempt to execute the global transaction will succeed, so the reservation phase (if successful) guarantees that the global transaction can *eventually* be executed successfully. That is, each of the subtransactions is guaranteed to eventually commit if it is re-executed whenever it aborts. The reservations made by a global transaction must be unreserved. An important (and in our opinion reasonable) assumption made, is that integrity constraint transaction failures are the only phenomenon that can cause a transaction that is repeatedly submitted to abort *indefinitely.* Therefore, the goal of reservation becomes that of ensuring that integrity constraint transaction failures cannot occur. This assumption is described in more detail below.

Persistent Failure Assumption. Many types of failures may occur in a distributed system, such as:

- **Integrity Constraint Transaction Failures.** Transaction failures that occur as a result of the database state.
- **Non-Integrity Constraint Transaction Failure.** For example, when the scheduler aborts a transaction due to a serialization conflict or deadlock.
- **System Failures.**
- **Media Failures.**
- **Communication Failures.**

We assume that only integrity constraint transaction failures can exist indefinitely.

For example, consider the case of a transaction that is trying to withdraw $100 from an account with under $100. This transaction could fail indefinitely, if no other transaction deposited more money into the account. If the other kinds of failures could exist indefinitely, any transaction management approach would seem to have difficulties. As a result, we assume that any transaction that is guaranteed to be free of integrity constraint transaction failures will eventually commit if it is re-executed.

For the above example, the reservation phase would consist of checking first to make sure there is at least $100 in the account, and then ensuring that any future transactions that would make the account go below $100 would be aborted. Unreservation would, in effect, remove the restriction that the account could go below $100. Later, we will discuss how we propose to actually implement this protocol in multidatabase systems.

2.1 Reservation Commitment Protocol

The reservable transaction model uses a commitment protocol that uses the semantics of the objects involved to provide atomic commitment with a lower degree of blocking than the traditional two phase commit protocol.

The key points of the reservation phase are that:

- The reservation ensures that the transaction will eventually be able to commit in the future, i.e. it will be free from integrity constraint transaction failures.
- The reservation will not cause database consistency to be violated.
- The reservation phase must be unreservable. That is, one must be able to undo the effects of the reservation.

Figure 2 provides a pictorial representation of the reservation commitment protocol. When a transaction is initiated by a user, the reservation phase is started. If all necessary reservations succeed, then the execution phase begins, otherwise, an unreservation phase begins where all successful reservation are undone. In the execution phase, the subtransactions are executed (and reservations are undone) as many times as is necessary until all of them successfully commit. Note that unreservations must be done as long as at least one of the reservations succeeds. The execution phase in this section is considered to include an unreservation phase. This is done to emphasize that the reservation commitment protocol is a two phase protocol as far as communication between the coordinator and participants is concerned.

In some sense, the reservation commitment protocol can be viewed as a generalization of the two phase commit protocol. In the extreme case, the reservation phase would involve

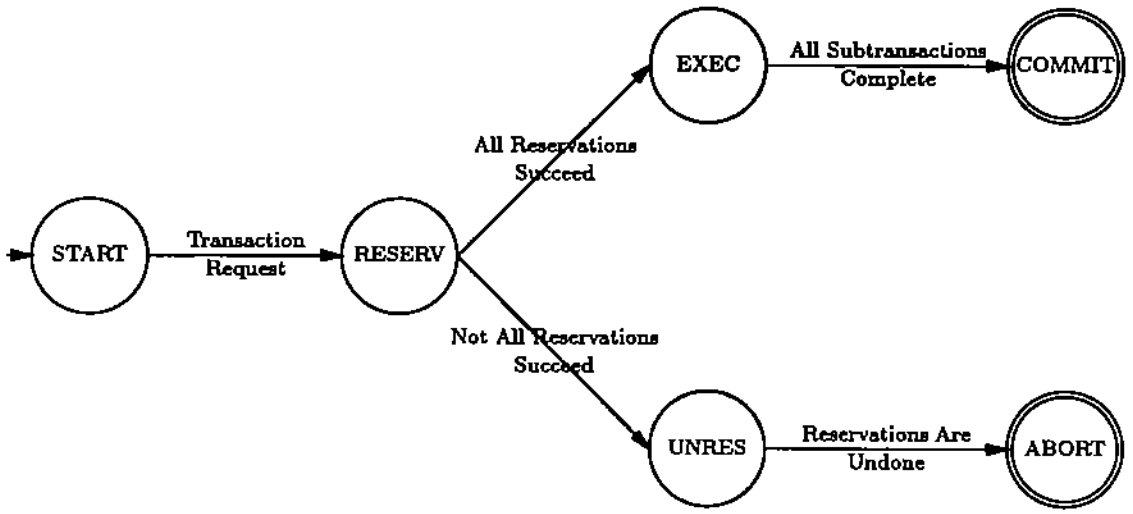

Figure 2: Reservation Commitment Protocol.

completely locking the data. This method should be effective for any types of objects, however, then the reservation method will have the same blocking behavior of two phase commit. Fortunately, however, one can often use the semantics of objects to reserve the subtransaction without completely locking the object involved. For example, as described earlier, if one wants to reserve a transaction that withdraws $100 from an account, one only needs to make sure that the account is "locked" to transactions that will cause it to go below $100; other transactions may still access the data.

Certain optimizations are possible for the commitment protocol. For example, it may be that the conditions of the reservation phase are implicitly satisfied, thus eliminating the need for this phase for some subtransactions. A subtransaction that withdraws all the money in an account, regardless of how much money is in the account, could be such an example. The case where *all* transactions have an implicit reservation corresponds to the unilateral commitment case in [HS91]. Also, if there is only one subtransaction, no reservation phase is necessary. More importantly, if only one of multiple subtransactions needs to be reserved, the reservation can be skipped if the subtransaction can be executed first. That is, the subtransaction that needs to be reserved is executed. If it succeeds, the other subtransactions are executed. If it fails, the transaction is aborted, and the other subtransactions are not attempted.

2.2 Example Reservable Transactions

In this section, we will provide example reservable transactions.

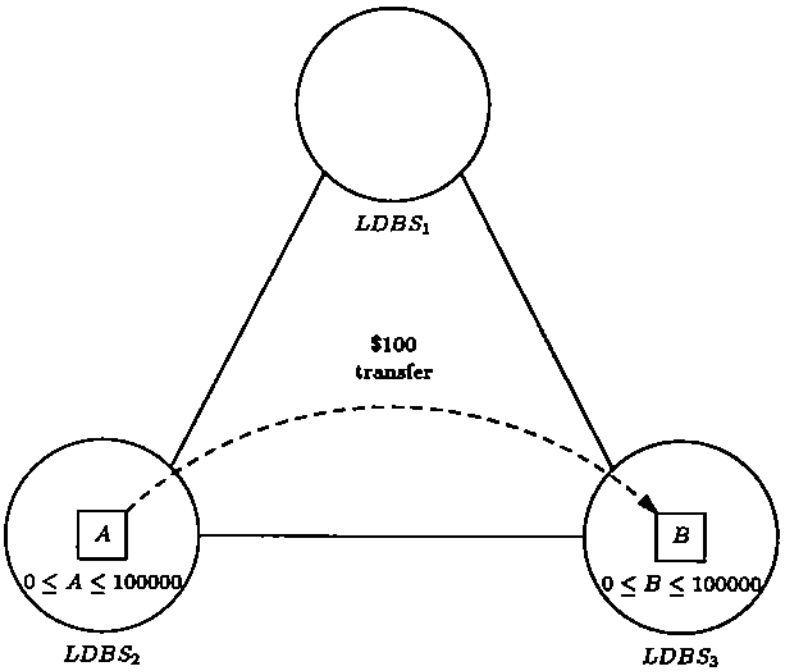

Figure 3: Transfer Example.

Bank Account Transfer Example. In this example the global transaction has two subtransactions, one that withdraws \$100 dollars from account A at $LDBS_2$, and another that deposits \$100 in account B at $LDBS_3$. So, in effect, the transaction is transferring \$100 dollars from one account to another. Suppose that account A cannot have a negative amount, and that similarly there is an upper limit (say \$100,000) on the amount in account B (possibly for insurance purposes). (See Figure 3).

The two subtransactions might look as follows:

```
subtransaction LDBS2:withdraw {
   amount := read(A);
   write(A, amount - 100);
   commit;
}

subtransaction LDBS3:deposit {
   amount := read(B);
   write(B, amount + 100);
   commit;
```

```
}
```

To reserve transactions, two data items *lower_limit* and *upper_limit* can be used. No transaction will be allowed to withdraw from an account past the lower limit, and no transaction will be allowed to deposit past the upper limit. Therefore, the account will always have at least *lower_limit* dollars, and at most *upper_limit* dollars. The lower limit will not be allowed to go above the upper limit. So, the reservation for subtransaction $LDBS_2$:withdraw might look as follows:

```
reservation LDBS2:withdraw {
    lower_limit := read(A_lower_limit);
    upper_limit := read(A_upper_limit);
    if (lower_limit + 100 < upper_limit) {
        abort;
    }
    else {
        write(A_lower_limit, lower_limit + 100);
        commit;
    }
}
```

That is, if possible, the reservation raises the lower limit (the limit past which transactions may not lower the account) to reserve the withdraw transaction. The corresponding unreservation would look as follows:

```
unreservation LDBS2:withdraw {
    lower_limit := read(A_lower_limit);
    write(A_lower_limit, lower_limit - 100);
    commit;
}
```

The reservation and unreservation for the deposit subtransaction will be as follows:

```
reservation LDBS3:deposit {
    lower_limit := read(B_lower_limit);
    upper_limit := read(B_upper_limit);
    if (upper_limit - 100 < lower_limit) {
        abort;
    }
```

```
    else {
        write(B_lower_limit, upper_limit - 100);
        commit;
    }
}

unreservation LDBS3:deposit {
    upper_limit := read(B_upper_limit);
    write(B_upper_limit, upper_limit + 100);
    commit;
}
```

Local transaction would be required not to make accounts go below their lower limit or above their upper limit. That is, local transactions that would invalidate a reservation must abort. For example, a local transaction at $LDBS_2$ that would deposit \$20 in account A would look as follows:

```
localtransaction deposit {
    lower_limit := read(A_lower_limit);
    if (lower_limit < 20) {
        abort;
    else {
        amount := read(A);
        write(A, amount - 20);
        commit;
    }
}
```

Airline Ticket Purchase Example. Another example might involve airline tickets. One might want to execute a global transaction that rents a hotel room from one database system, and purchases airline tickets from another database system. A reservation transaction for the airline tickets could simply lock the number of tickets needed. For flight X, $X_{not\_purchased}$ might represent the number of tickets left that have not been purchased. And X_{number_left} could be used to indicate the maximum tickets that could be purchased without violating an existing reservation. That is:

$$X_{number\_left} = X_{not\_purchased} - (number\_of\_purchases\_reserved)$$

The reservation and unreservation transaction programs to reserve and unreserve "number" ticket purchases for flight X might look as follows:

```
reservation LDBS_j:tickets( number ) {
    number_left := read(X_number_left);
    if (number > number_left) {
        abort;
    }
    else {
        write(X_number_left, number_left - number);
        commit;
    }
}

unreservation LDBS_j:tickets( number ) {
    number_left := read(X_number_left);
    write(X_number_left, number_left + number);
    commit;
}
```

The reservation could become more complex if factors such as seat position (i.e. window, middle, aisle), and relative position were considered. For example, suppose a reservation transaction wanted to reserve three adjacent seats (i.e. 1 window, 1 middle, and 1 aisle). All transaction that left less than 3 available seats would clearly have to be aborted, but also other transactions, such as a transaction that tried to lock all remaining aisle seats.

Generic Reservable Transaction Example. A generic method for implementing reservable transactions is to completely lock the data in the reservation stage. For any data item(s), one data item can be used as the lock, which will have two values: "locked" and "unlocked". Local and reservation transactions abort if the lock is set. Reservation transactions set the lock if it is not already set, and the data value(s) in the database will not cause the execution to abort. The unreservation transaction releases the lock. In general, this method should work regardless of the semantics of the data item(s) involved, however it exhibits a similar blocking behavior to two phase commit. The reservation and unreservation transactions could be as follows:

```
reservation LDBS_j:generic {
    lock := read(X_lock);
```

```
    if (lock == "locked") {
        abort;
    }
    else {
        write(X_lock, "locked");
        commit;
    }
}

unreservation LDBS_j:generic {
    write(X_lock, "unlocked");
    commit;
}
```

2.3 Transaction Method Comparison

In this section we will compare reservable transactions to traditional transactions, and to compensatable transactions.

Consider the bank account transfer example in Section 2.2. To reserve the transfer transaction T, it is necessary to make sure that account A has at least \$100 and account B has at most \$99,900, and that this condition will not change until after T commits. Once this has been done, it can be guaranteed that transaction T can execute without integrity constraint transaction failures.

Assuming that the reservation phase is successful, then the two subtransactions of the global transaction (the one that withdraws \$100 from A, and the one that deposits \$100 in account B) will be re-executed as many times as is necessary until they successfully commit. Local transactions will still be able to access account A and B after the reservation phase.

For this example, the traditional method may result in blocking, and the compensating method cannot provide correctness.

Reservation vs. Traditional Transaction Management. If standard two phase commitment was used, then after the first phase if the coordinator ($LDBS_1$) failed, then other transactions could be blocked from accessing A and B until $LDBS_1$ has recovered. And, in any event, other transactions would be blocked until the coordinator responds. Whereas, using reservation commitment, if $LDBS_1$ failed after the reservation phase, accounts A and B could still be accessed by local transactions running at $LDBS_2$ and $LDBS_3$. (See Ap-

pendix A, Figure 6 for the transaction program corresponding to the traditional case.)

Reservation vs. Compensation. In the compensating method (see Appendix A, Figure 7) correctness cannot be preserved since local transactions may see an incorrect database. Consider the case where a local transaction that pays interest on account A and sees A after a withdrawal that will later be compensated. For example, suppose that the account initially has \$100, and then a withdraw transaction that withdraws \$100 is executed that will later be compensated. Then suppose a local transaction makes an interest payment of 1% before the compensation is performed (i.e. when the account has \$0). After the compensation is performed the account will have \$100 when it should have \$101, since the interest transaction saw an inconsistent state. In this case, interest on the account will, in effect, be lost, and therefore database consistency will not be preserved. This example is covered in more detail in [MEK92]. In general, compensation is not applicable when non-commutative operations can be performed on an object, or when the execution of a subtransaction corresponds to some event which cannot be compensated.

Another point to notice is that it is possible that compensation could require a reservation phase. Suppose, for example that interest payments are not allowed on accounts, and only withdraws and deposits (which are commutative operations) are allowed. Suppose also that there is \$0 in account B and that the transfer transaction deposits \$100 in B. The compensation of this transaction would be to withdraw \$100 from B. However, suppose another transaction withdraws this money before the compensation can be performed. The result would be that the compensation could fail indefinitely. So, reservation could be used to guarantee the compensation can eventually commit by making sure the account does not go below \$100. This example demonstrates that reservation commitment can be used to help support compensation.

There are cases, however, where compensation is more appropriate than reservation. It may be that the semantics of the object do not allow reservation, other than one that causes blocking to be done, but compensation can be performed. In summary, we do not see reservation as something to be used in place of compensation, but in conjunction with it. In this paper, however, we concentrate on presenting the reservation approach, so we do not present a detailed description of a method that integrates both approaches. The comparison of reservation to two phase commit and compensation is summarized in Table 1.

Other Methods. The approach presented in [HS91] also re-executes subtransactions until all of them commit. The unilateral commit approach, however, assumes that "the decomposed execution (of a multi-site transaction) does not compromise database integrity" [HS91].

	Commitment Method		
	2PC	Reservation	Compensation
Local Transactions Blocked Until Coordinator Responds	YES	MAYBE[1]	NO
Global Transactions Blocked Until Coordinator Responds	YES	MAYBE[2]	NO
Operations Must Be Commutative	NO	NO	YES
Works When Transaction Corresponds to Irreversible Physical Event	YES	YES	NO

[1] Depends on the semantics of the object involved. Can range from total blocking to partial or no blocking.

[2] Global transactions that access only one system, or that access only the one system in common (with the global transaction that has not completed), would not have to be blocked.

Table 1: Comparison of Commitment Approaches

Or, in our terminology, subtransactions are assumed to be free from possible integrity constraint transaction failures. So, the unilateral commitment method can be viewed as the special case of our reservation commitment method where all the subtransactions are implicitly reserved (i.e. automatically free from integrity constraint transaction failures).

In [EJK91] another approach is presented where subtransactions are re-executed until all of them commit. In this approach, one of the subtransactions may possibly be subject to integrity constraint transaction failures, as long as the data dependencies of the global transaction allow it to be executed first. If one of the subtransactions of a global transaction is subject to integrity transaction failures, it is executed first. If it succeeds then the other subtransactions are executed and re-executed until they commit, otherwise the global transaction aborts. Again, this approach can be viewed as a special case of the reservable transaction approach.

An approach for reliable multidatabase transaction management is presented in [BST90]. This approach also attempts to re-execute subtransactions until they commit. In this approach, eventual commitment is not simply assumed as in unilateral commitment. The data in the database is partitioned into globally updatable and locally updatable sets. This partitioning will prevent local transactions from modifying the globally updatable data in undesirable ways.

Although it was developed for more tightly coupled distributed database systems and does not deal with the issues of local system autonomy, the Escrow Transactional Method [O'N86] is similar to our method in that it uses reservation-like approach. The Escrow method permits record updates by transactions without forbidding simultaneous access. Each transaction must test an "Escrow-type" data item before the transaction attempts to perform an "Escrow-type update" to the item. The method is similar to our reservation approach, however, the operations performed on the data item must be commutative, so that the updates to "Escrow-type" data items can be committed in any order. So, the Escrow approach has the same limitation as the compensation approach in regard to operation properties. Our reservation approach does not make this kind of assumption and still allows for non-blocking commitment of global transaction. This feature of our approach is demonstrated in the previous bank account transfer example.

3 Formal Model of Reservable Transactions

In this section we will present a formal model for reservable transactions.

3.1 Multidatabase System Model

Our formal model of a multidatabase system consists of:

1. a set of n local database systems $\{LDBS_1, LDBS_2, ..., LDBS_n\}$, where each $LDBS_i$ is considered to have exactly one database LDB_i.
2. a set of local transactions $\bigcup_{i=1}^{n}\{L_{1,i}, L_{2,i}, ..., L_{q_i,i}\}$.
3. a set of reservable transactions $\{\mathcal{R}_1, \mathcal{R}_2, ..., \mathcal{R}_p\}$
4. a set of n local histories $\{H_1, H_2, ..., H_n\}$
5. a global history H.

Basically, our definitions for transactions and histories are extensions of those found in [BHG87].

Definition 3.1 (Local Database) *A local database LDB_i is considered to be a set of objects (or data items) $\mathcal{D}_i$.*

Definition 3.2 (Single System Transaction) *A single system transaction $T_{i,j}$ is a partial order with ordering relation $<_{T_{i,j}}$ where:*

1. $T_{i,j} \subseteq \{r_{T_{i,j}}[x], w_{T_{i,j}}[x] \mid x \in \mathcal{D}_j\} \cup \{a_{T_{i,j}}, c_{T_{i,j}}\}$
2. $a_{T_{i,j}} \in T_{i,j}$ *iff* $c_{T_{i,j}} \notin T_{i,j}$
3. *if* p *is* $c_{T_{i,j}}$ *or* $a_{T_{i,j}}$, *then* $\forall o(o \neq p), o <_{T_{i,j}} p$
4. *if* $r_{T_{i,j}}[x], w_{T_{i,j}}[x] \in T_{i,j}$, *then either* $r_{T_{i,j}}[x] <_{T_{i,j}} w_{T_{i,j}}[x]$ *or* $w_{T_{i,j}}[x] <_{T_{i,j}} r_{T_{i,j}}[x]$

(1) says that transactions consist of read, write, abort and commit operations, and that read and write operations operate only on data items at the system where the transaction executes. (2) says that a transaction will have either an abort or a commit operation, but not both. (3) the abort or commit operation will be the last operation. (4) says that the ordering relation will determine the order between operations that access the same data item. $r[x]$ implies a read of data item x. $w[x]$ implies writing to data item x. And, $w[x, v]$ will be used to denote writing value v to data item x when it is necessary to make reference to the value written. For $T_{i,j}$, j specifies the system of $T_{i,j}$. Where j can be derived from context, T_i may be used. In general, o will be used to represent any possible operation (i.e. r, w, c, and a), and $o[x]$ will be used to represent any operation that is performed on a data item (i.e. $r[x]$, and $w[x]$).

Definition 3.3 (Local Transaction) *A local transaction is a single system transaction that is submitted directly to an LDBS, and that preserves database consistency if executed in isolation and to completion.*

Definition 3.4 (Global Transaction) *A global transaction G_i consists of a set of single system transactions that are submitted by the GTM, and has the following properties:*

1. $G_i = \{T_{i,j_1}, T_{i,j_2}, ..., T_{i,j_p}\}$, *where* $1 \leq j_k \leq n$, *for* $1 \leq k \leq p$
2. *if* $w[x, value] \in T_{i,j}$ *then* $value = f(y_1, y_2, ..., y_n)$ *where for* y_l, $1 \leq l \leq n$, *either:*
   (a) $y_l \in \mathcal{D}_j$, $r_{T_{i,j}}[y_l] \in T_{i,j}$, $r_{T_{i,j}}[y_l] <_{T_{i,j}} w_{T_{i,j}}[x, value]$, *or,*
   (b) $y_l \in \mathcal{D}_k$ $(k \neq j)$, $r[y_l] \in T_{i,k}$, $c_{T_{i,k}} <_{G_i} c_{T_{i,j}}$
3. *preserves database consistency if executed in isolation and to completion.*

(1) says that a global transaction consists of a set of single system transactions (which are called subtransactions) that execute at some subset of the LDBSs in the multidatabase. (2) says that each value written by a subtransaction $T_{i,j}$ will be a function of values previously read by that subtransaction, or of the values read by another subtransaction belonging to the same global transaction that commits before subtransaction $T_{i,j}$.

Next we will define reservable transactions. A reservable transaction is actually a set of global transactions, so it is not a transaction in the sense that it does not support the ACID properties [HR83]. Specifically, it does not not support isolation, since other transactions could see the results of its committed global transactions, even if not all of its global transactions have committed. However, a reservable transaction is considered to be a logical unit of work, and will preserve database consistency, so we use the term transaction.

Note that each unreservation transactions could be included within its corresponding standard subtransaction, but this is not done here because it would make the formalization more difficult to present.

Definition 3.5 (Reservable Transaction) *A reservable transaction $\mathcal{R}_i$ consists of a set of global transactions. It has the following properties:*

1. $\mathcal{R}_i = \{S_i\} \bigcup \{R_{i,j_1}\} \cup \{R_{i,j_2}\} \cup .... \cup \{R_{i,j_p}\} \bigcup \{U_{i,j_1}\} \cup \{U_{i,j_2}\} \cup ... \cup \{U_{i,j_p}\}$,
   $S_i = \{S_{i,j_1}, S_{i,j_2}, ..., S_{i,j_p}\}$, *where* $1 \leq j_k \leq n$, *for* $1 \leq k \leq p$
2. $o_{S_{i,j}} \in S_{i,j}$ *iff* $c_{R_{i,k}} \in R_{i,k}$, $\forall k$ *such that* $R_{i,k} \in \mathcal{R}_i$

3. *if* $o_{S_{i,j}} \in S_{i,j}$ *then* $c_{R_{i,k}} <_{\mathcal{R}_i} o_{S_{i,j}}, \forall j, k$

4. $o_{U_{i,j}} \in U_{i,j}$ *iff* $c_{R_{i,j}} \in R_{i,j}$

5. *if* $o_{U_{i,j}} \in U_{i,j}$ *and* $c_{R_{i,j}} \in R_{i,j}$, *then* $c_{R_{i,j}} <_{\mathcal{R}_i} o_{U_{i,j}}$

6. *if* $c_{R_{i,k}} \in R_{i,k}$, $\forall k$ *such that* $R_{i,k} \in \mathcal{R}_i$ *then* $c_{S_{i,j}} <_{\mathcal{R}_i} o_{U_{i,j}}$

So, $\mathcal{R}_i$ consists of $(2p+1)$ global transactions: 1 standard transaction, p reservation transactions, and p unreservation transactions. S_i consists of p standard subtransactions. (2) and (3) say that no standard subtransaction of a global transaction will be executed until all of its reservations have committed. (4) and (5) say that an unreservation will be executed if and only if its corresponding reservation has committed. (6) says that if all the reservations are successful, an unreservation transaction cannot be executed until its corresponding standard subtransaction has committed.

Definition 3.6 (Standard Subtransaction) *A standard subtransaction $S_{i,j}$ is a single system transaction that is part of the execution phase of a reservable transaction.*

Definition 3.7 (Reservation Transaction) *A reservation transaction $R_{i,j}$ for subtransaction $S_{i,j}$ is a single system global transaction, and must have the following property:*

> *the commitment of $R_{i,j}$ guarantees that $S_{i,j}$ and $U_{i,j}$ will execute free from integrity constraint transaction failures. That is, the state of the data in the database $\mathcal{D}_j$ will not cause $S_{i,j}$ or $U_{i,j}$ to abort.*

Definition 3.8 (Unreservation Transaction) *An unreservation transaction $U_{i,j}$, for standard subtransaction $S_{i,j}$ and reservation transaction $R_{i,j}$ is a single system transaction, and must have the following properties:*

1. *$U_{i,j}$ undoes the effects of $R_{i,j}$. That is, it removes the guarantee that $S_{i,j}$ and $U_{i,j}$ will execute free from integrity constraint transaction failures.*

2. *It will leave the database in the same state as if $R_{i,j}$ and $U_{i,j}$ had not committed. Specifically, if one has operations accessing $\mathcal{D}_j$ that include the operations of $R_{i,j}$ and $U_{i,j}$:*

   $o_1\ o_2\ \ldots\ o_i\ R_{i,j}\ o_{i+1}\ o_{i+2}\ \ldots\ o_j\ U_{i,j}\ o_{j+1}\ o_{j+2}\ \ldots\ o_k$

   then the state of $\mathcal{D}_j$ should end up being the same as if these same operations without those of $R_{i,j}$ and $U_{i,j}$ executed:

$o_1\ o_2\ \ldots\ o_i o_{i+1}\ o_{i+2}\ \ldots\ o_j\ o_{j+1}\ o_{j+2}\ \ldots\ o_k$

Definition 3.9 (Local History) *The local history H_i is a partial order with ordering relation $<_{H_i}$ where:*

1. $H_i = (\bigcup_{j=1}^{m} L_{j,i}) \cup \{G_{j,i} | G_{j,i} \in G_j, \text{ for } j = 1 \text{ to } p\}$
2. $<_{H_i} \supseteq (\bigcup_{j=1}^{m} <_{L_{j,i}}) \cup \{<_{G_{j,i}} | G_{j,i} \in G_j, \text{ for } j = 1 \text{ to } p\}$
3. *for any two conflicting operations $p, q \in H_i$, either $p <_{H_i} q$ or $q <_{H_i} p$.*

(1) says that the local history includes the operations of those local transactions and global subtransactions submitted to $LDBS_i$. (2) says that the history ordering relation includes all the orderings in the transactions whose operations are included. (3) says that the ordering of every pair of conflicting operations is determined by $<_{H_i}$.

Definition 3.10 (Global History) *The global history H is a partial order with ordering relation $<_H$ where:*

1. $H = \bigcup_{i=1}^{n} H_i$,
2. $<_H \supseteq (\bigcup_{i=1}^{n} <_{H_i}) \cup (\bigcup_{i=1}^{p} <_{G_i})$

(1) says that the operations in H consists of the operations in all the local histories. (2) says that the global history includes all the orderings of the local histories as well as those between global subtransactions belonging to the same global transaction.

3.2 Transaction Management Requirements

In this section we will describe the properties that the transaction management system must possess in order to support the correct execution of reservable transactions.

Local Database Systems. LDBSs must produce histories that are strongly recoverable [BGRS91] (which implies serializability and recoverability). Formally stated:

$H_i \in SRC$, for $1 \leq i \leq n$, $SRC \subset RC$, $SRC \subset SR$

This requirement is needed so that serializable global histories may be produced.

Global Transaction Manager. The GTM must provide the following:

1. **Consistent Commitment Order.** The commitment order between different subtransactions for any pair of global transactions, must be consistent. Formally stated:

   if $c_{G_{i,a}} < c_{G_{j,a}}$, then $\forall LDBS_b (a \neq b)$ where $G_{i,b} \in G_i$ and $G_{j,b} \in G_j$, $c_{G_{i,b}} <_H c_{G_{j,b}}$

2. **Reservation Commitment Protocol.** The GTM will support the reservation commitment protocol. The GTM will execute reservation and unreservation transactions, and standard subtransactions in the correct order. And, the GTM will handle the re-execution of standard subtransactions and unreservations.

(1) is required so that serializable global histories may be produced.

3.3 Correctness of Reservable Transactions

The meaning of correct execution is complicated in the multidatabase model due to the fact that a global transaction may have some subtransactions that are committed and some that are aborted. However, in any event, database consistency must be preserved. Since we assume that transactions preserve database consistency, we define correct execution in the multidatabase environment as follows.

Definition 3.11 (Global History Correctness) *A global history H is considered to be correct if it is serializable, recoverable, and has no partial executions of transactions.*

That is, since each transaction preserves database consistency, if the transactions are executed as if they are executed to completion, and in isolation, database consistency will be preserved.

Lemma 3.1 *All transactions will either execute completely, or not at all.*

Proof: There are exactly three cases where a transaction would not execute completely:

Case 1: A local transaction commits part of its effects.

Case 2: A global subtransaction commits part of its effects.

Case 3: A global transaction commits part of its subtransactions.

(Case 1–2) Since each LDBS is required to produce schedules which are are serializable and recoverable, these cases cannot occur.

(Case 3) This case cannot occur for the following reason. First of all, since standard transactions are the only global transactions with multiple subtransactions, if this case could occur, it would have to occur for a standard transaction. By definition of reservable transactions, if one subtransaction of a standard transaction has committed, then all the reservation transactions of the reservable transaction must have committed previously. Therefore, by definition of reservable transactions, all standard subtransactions can execute free from integrity constraint transaction failures. From our persistent failure assumption, it follows that the standard subtransactions will eventually commit if re-executed. Since the GTM requirements are such that the GTM must keep re-executing the standard subtransactions until they all commit, then eventually the entire global standard transaction will be committed. □

Lemma 3.2 *Global histories produced by the reservation transaction management method will be recoverable.*

Proof: The global history H will be recoverable if whenever transaction T_i reads from T_j $(i \neq j)$ in H and $c_i \in H$, $c_j <_H c_i$. That is, no transaction reads the value of a data item last written by a transaction that has not committed. Only the following cases are possible:

1. $T_i = L_{a,c}$ and $T_j = L_{b,c}$ $(a \neq b)$. (Two different local transactions at the same $LDBS$.)
2. $T_i = L_{a,c}$ and $T_j = G_{x,c}$. (a local transaction and a global subtransaction at the same $LDBS$.)
3. $T_i = G_{x,c}$ and $T_j = L_{a,c}$. (A global subtransaction and a local transaction at the same $LDBS$.)
4. $T_i = G_{x,a}$ and $T_j = G_{y,a}(x \neq y)$. (Two global subtransactions at the same $LDBS$ that belong to different global transactions.)
5. $T_i = G_{x,a}$ and $T_j = G_{x,b}(a \neq b)$. (Two global transactions at different $LDBSs$ that belong to the same global transaction.)

Note that by definition the case of $T_i = L_a, T_j = L_b(a \neq b)$ is not possible.

(Cases 1–4) Since each LDBS is required to produce recoverable histories, these cases cannot occur.

(Case 5) By definition of global transactions, a global subtransaction may only read from global subtransactions that belong to the same global transaction if they have committed. □

Lemma 3.3 *Global histories produced by the reservation transaction management method will be serializable.*

Proof: The global transaction manager will control the commit order of the global subtransactions to make sure their relative order is the same at each site. Since a consistent commitment is provided and we assume that each LDBMS produces strongly recoverable executions, it can be deduced from [BGRS91]. □

Theorem 3.1 *Global histories produced by the reservation transaction management method will be correct.*

Proof: Since the reservation transaction management method will execute all transaction either completely, or not at all (Lemma 3.1), and since the global histories produced will be recoverable (Lemma 3.2) and serializable (Lemma 3.3), correctness will be maintained. □

4 Implementation of Reservable Transactions

4.1 Transaction Management

To implement reservation transaction management, it is necessary for the multidatabase system to provide the properties discussed in section 3.2. Each LDBS should ensure that its histories are serializable and strongly recoverable. The GTM must ensure that global transactions have a consistent commitment order and that the reservation commitment protocol is followed.

Basically, the LDBSs are assumed to provide the properties described. An LDBS that supports the traditional transaction concept and uses strict two phase locking for its concurrency control method would provide the necessary properties.

The GTM needs to provide quasi serializable executions [DE89], that is, executions with a consistent commitment order. One simple method for providing a consistent commitment order between global transactions, is for the GTM to use global locking to coordinate the executions of global transactions. Before the execution of a global transaction, each site that the transaction will access should be locked first by the GTM. A locked site cannot process other global subtransactions, but can process local transactions. If all the global lock requests have been granted from GTM, the global transaction can be submitted to local databases to execute. The global lock on a site can be released only after the subtransaction that accesses the site successfully commits.

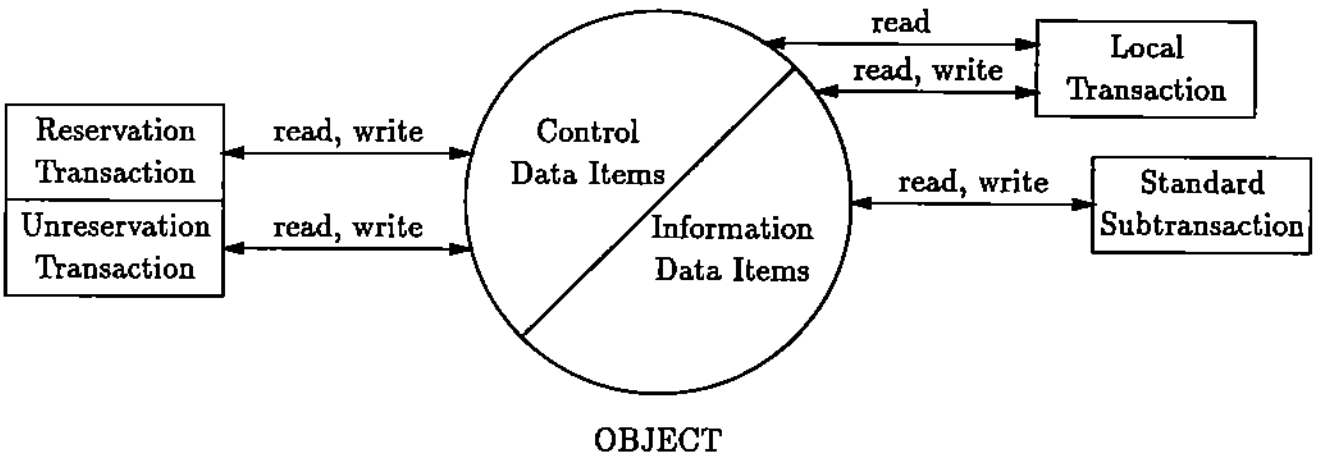

Figure 4: Data Partitioning.

The GTM also needs to implement the reservation commitment protocol. This is relatively straightforward. The GTM must simply execute the global transactions of a reservable transaction in the correct order, and re-execute failed standard subtransactions and unreservation transactions.

4.2 Constructing Reservable Transactions

Example reservable transactions have been presented in Section 2.2. In this section we attempt to provide insight into the construction of reservable transactions.

The reservation phase of a reservable transaction must prevent transactions from committing that might cause integrity constraint transaction failures for the standard subtransactions. As was seen in the transfer example, this did *not* require locking the data completely. A lower limit and upper limit were used to block only appropriate transactions.

In general it seems to be useful to think of the database as consisting of objects where each object has *control data items* and *information data items*. Reservation and unreservation transactions only access the control data items. Standard subtransactions only access the information data items. Local transactions only read and write the information data items, with the exception that they may read the control data items to decide if they should abort.

For each operation to be performed on the object that requires reservation, four corresponding operations would exist:

1. A local version of the operation that checks the control data to see if it might possibly

invalidate a reservation.

2. A global version.

3. A global version reservation.

4. A global version unreservation.

The examples in Section 2.2 conform to this approach. For example, in the bank account transfer example, A and B are information data items, and A_{lower_limit}, A_{upper_limit}, B_{lower_limit}, and B_{upper_limit} are control data items. To further illustrate this approach, we show how the account object might be defined using C++ syntax:

```
class Account {
private:
   int balance, upper_limit, lower_limit;
public:
   Account();

   int Read();

   int Deposit(int amount);
   void GlobalDeposit(int amount);
   int ReserveGlobalDeposit(int amount);
   void UnreserveGlobalDeposit(int amount);

   int Withdraw(int amount);
   void GlobalWithdraw(int amount);
   int ReserveGlobalWithdraw(int amount);
   void UnreserveGlobalWithdraw(int amount);

};
```

This says that the object Account has three private (essentially, they can only be accessed by code in the object's methods) variables: *balance*, *upper_limit*, and *lower_limit*. *balance* reflects the amount of the account, and *lower_limit* represents an amount the account cannot go below, and *upper_limit* represents an amount the account cannot go above. *Read()* returns the balance of the account and does not require a reservation.

4.3 Effects on Local Autonomy

One of the crucial properties of multidatabase algorithms to examine is their effect on local system autonomy. Reservable transactions help to preserve execution autonomy. That is, they reduce the blocking of data at local systems. Furthermore, implementing reservable transaction in pre-existing systems does not require the code of the LDBS to be modified. This is a big advantage, since changing the LDBS code may be very expensive, if not practically impossible (suppose one only has the executable code, and not the source code, for example). However, local transactions will have to be modified to follow the reservation commitment protocol. Although, this only needs to be done for local transactions that *modify* data that is globally available. In addition, additional data (e.g. lower_limit and upper_limit in the example) may need to be defined. In summary, while reservable transaction do not completely preserve local system autonomy, they can provide a high level of execution autonomy, and do not require the LDBS code to be modified.

5 Conclusions

In this paper we have presented reservable transactions. Reservable transactions have a reservation phase that attempts to guarantee that the transaction can eventually commit. The reservation phase utilizes data/transaction semantics to reduce the blocking and long delays of traditional transaction management methods such as two phase commit.

The compensation approach also utilizes data/transaction semantics to avoid locking and long delays. However, compensation essentially takes an optimistic approach, whereas reservable transactions essentially take a pessimistic approach. More importantly, reservable transactions can handle cases where compensation is not appropriate. For example, in cases where the operations performed on some data are not commutative (i.e. the order of operations is important), or where transactions correspond to actual physical events that may be hard to compensate (e.g. launching a missile or releasing money from an automatic teller).

In certain cases where compensation can be performed, however, compensation could have a lower degree of blocking than reservation because of the particular data/transaction semantics involved. Also, the reservation approach of reservable transactions can be used to increase the cases where compensation is appropriate. This is because it is possible for the compensating transaction to fail (indefinitely) in certain cases. Reservation can be used to ensure that the compensating transaction will eventually commit. So, in regard to compensation, we see reservation as an approach that could be used in conjunction with

compensation, as opposed to being used as a replacement.

References

[BGRS91] Y. Breitbart, D. Georgakopolous, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 9, 1991.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems.* Addison-Wesley Publishing Co., 1987.

[BST90] Y. Breitbart, A. Silberschatz, and G. Thompson. Reliable transaction management in a multidatabase system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 215–224, May 1990.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, pages 347–355, Amsterdam, The Netherlands, August 1989.

[EJK91] A. Elmagarmid, J. Jing, and W. Kim. Global commitment in multidatabase systems, 1991. (submitted for publication, also available as Purdue University Technical Report CSD-TR-91-017).

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM Conference on Management of Data*, pages 249–259, May 1987.

[Gra78] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, pages 624–633. Springer-Verlag, Berlin, 1978.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), July 1983.

[HS91] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the 7th Intl. Conf. on Data Engineering*, pages 296–304, Kobe, Japan, April 1991.

[LS76] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[MEK92] James G. Mullen, Ahmed K. Elmagarmid, and Won Kim. On the impossibility of atomic commitment in multidatabase systems. In *Proceedings Of The International Conference on Systems Integration*, June 1992. (to appear).

[O'N86] P. E. O'Neil. The escrow transaction method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

Appendix

A

	WITHDRAW	DEPOSIT

RESERVATION PHASE	**reservation** $LDBS_2$:withdraw {
	lower_limit := read(A_{lower_limit});
	upper_limit := read(A_{upper_limit});
	if (lower_limit + 100 < upper_limit) {
	abort;
	}
	else {
	write(A_{lower_limit}, lower_limit + 100);
	commit;
	}
	}
		reservation $LDBS_3$:deposit {
		lower_limit := read(B_{lower_limit});
		upper_limit := read(B_{upper_limit});
		if (upper_limit - 100 < lower_limit) {
		abort;
		}
		else {
		write(B_{upper_limit}, upper_limit - 100);
		commit;
		}
		}

EXECUTION PHASE	**subtransaction** $LDBS_2$:withdraw {
	amount := read(A);
	write(A, amount - 100);
	commit;
	}
		subtransaction $LDBS_3$:deposit {
		amount := read(B);
		write(B, amount + 100);
		commit;
		}

UNRESERVATION PHASE	**unreservation** $LDBS_2$:withdraw {
	lower_limit := read(A_{lower_limit});
	write(A_{lower_limit}, lower_limit - 100);
	commit;
	}
		unreservation $LDBS_3$:deposit {
		upper_limit := read(B_{upper_limit});
		write(B_{upper_limit}, upper_limit + 100);
		commit;
		}

Figure 5: Reservable Transfer Transaction Example

	WITHDRAW	DEPOSIT

EXECUTION PHASE	**subtransaction** $LDBS_2$:withdraw {
	amount := read(A);
	if (amount < 100) {
	abort;;
	}
	else {
	write(A, amount - 100);
	commit;
	}
	}
		subtransaction $LDBS_3$:deposit {
		amount := read(B);
		if (amount > 99000) {
		abort;
		}
		else {
		write(B, amount + 100);
		commit;
		}
		}

Figure 6: Traditional Transfer Transaction Example

	WITHDRAW	DEPOSIT

EXECUTION PHASE	**subtransaction** $LDBS_2$:withdraw {
	amount := read(A);
	if (amount < 100) {
	abort;;
	}
	else {
	write(A, amount - 100);
	commit;
	}
	}
		subtransaction $LDBS_3$:deposit {
		amount := read(B);
		if (amount > 99000) {
		abort;
		}
		else {
		write(B, amount + 100);
		commit;
		}
		}

COMPENSATION PHASE	**compensation** $LDBS_2$:withdraw {
	amount := read(A);
	write(A, amount + 100);
	commit;
	}
		compensation $LDBS_3$:deposit {
		amount := read(B);
		write(B, amount - 100);
		commit;
		}

Figure 7: Compensation Transfer Transaction Example